LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-209020

build - A Directory Savvy Replacement for Make

J. F. Reus

January 14, 2005

NECDC 2004
Livermore, CA, United States
October 4, 2004 through October 7, 2004

Proceedings from the NECDC 2004

build
A Directory Savvy Replacement for Make (U)

Jim Reus

reus@llnl.gov

Lawrence Livermore National Laboratory, Livermore, CA 94551

The build utility is presented as an alternative to make. It is very portable and was designed with parallel operation from the start. Build extends the traditional make model by extending the include facility to allow for construction of dependency lattices that span directory trees. A number of features provided by build support improved meta-project component encapsulation giving improved project scalability. Build is generally compatible with most make implementations and supports the extensive macro function facility introduced by GNU make. (U)

Introduction

All but the most trivial software packages are generally constructed from multiple source files with many steps being required to generate lexer and parser source files, compile the numerous source files, assemble libraries from object files, and link object files and libraries to form the binary executable application. During the development process, the construction process may be repeated many times as source code is modified requiring the application to be rebuilt. To address this problem the original make tool was developed. However this tool was developed when applications were rather simple with a limited number of source files in single directories. Few header files outside of the standard system header files were used. The graph formed by the file-to-file dependencies was generally simple with little cross branching and limited depth.

The increased complexity of modern software systems has made the traditional make tool less capable of managing the problem. Good software development practices have led to more and more source files arranged in directory hierarchies. Modular development of the software has resulted in the proliferation of developer implemented header files and libraries that are part of the application. Code reuse and object oriented design has made the problem even worse by hiding the necessity to recompile source files. In short, the dependency graph has become much more complex, less a tree and more lattice-like. Keeping track of these dependencies rapidly becomes difficult with the increasing complexity of the software. In addition the shear volume of source code has increased the

need to avoid unnecessary compiles or relinks. The increased number of source files to be compiled has also increased the opportunity for parallelism but the increased complexity of the directory hierarchy and file-file dependencies has at the same time made it harder to exploit parallelism.

This paper gives a brief introduction to *build* – an improved implementation of make. A number of common issues are presented that contrast how build and make are used. In addition a number of new features provided by build are described. This presentation is by no means a complete description of build; a separate document is available that serves as a reference and user's guide.

Build

Build is a tool intended as an alternative to make. It is a compatible replacement for most make implementations and is a near superset of GNU make only lacking in a few of GNU make's special macro functions. One of the core requirements in the design of build was compatibility with make. In addition to the facilities provided by most make implementations build also:

- Understands projects with source in hierarchical directory trees.
- Implements a rich set of relationships such as:
  - Dependency
  - Inclusion
  - Copy
- Properly supports parallel builds.
- Provides improved pattern rules (along with old style make suffix rules).
- Supports multi-target rules.
- Supplies a number of additional directives dealing with complex dependency lattices and multiple products (goal objects).

In addition build is also:

- Scalable
- Compatible
- Fast and widely portable.

Note that since build is for the most part a superset of make, it will accept most well written makefiles. In fact, if not told to use a specific input file, build will first look for a file named *Buildfile* (or *buildfile*) and if that is not found will look for *Makefile* (or *makefile*).

Directory hierarchies

As projects become increasingly complex, it is a good software engineering practice to divide-up the multitude of source files into modules, one per directory, forming a directory hierarchy. Build provides a number of features to deal properly with directory hierarchies. At one point in its early development it was expected that a single buildfile would manage a project even if it spanned multiple directories. Build simply understood the relative pathnames in the rules. Such a scheme becomes rapidly unmanageable as the number of source files increases and the directory structure grows. Users of make generally use *recursive makes* – makefiles that contain rules that invoke make in the subdirectory. However each invocation of make has only a local view; no one invocation has the big picture. The only information flowing from make to make is the request to make some target, and the success or failure of the request. This limitation of information flow leads to unnecessary work and sometimes to missed work.

To get the big picture build supports an extended include mechanism to permit each directory to have its own local buildfile dealing only with those issues local to the directory. Build uses a new form of the familiar make-style include directive to collect the various buildfiles. Build recognizes the position of each buildfile in the directory hierarchy and so can *relocate* the contents of each buildfile appropriately. The result is a global view assembled from buildfiles developed locally.

The difference between how one handles directory hierarchy using make and build:

- When using make the developer generally uses either a grand makefile that understands the whole directory hierarchy (unmaintainable) or rules are added to parent directories that invoke make in the subdirectory using a makefile local to the subdirectory. For example:

  makefile

  ```
  all       :
            cd subdir1 ; make $@
            cd subdir2 ; make $@

  clean     :
            cd subdir1 ; make $@
            cd subdir2 ; make $@

  install   :
            cd subdir1 ; make $@
            cd subdir2 ; make $@
  ```

  The trouble with recursive makes is that there is little information exchange between the parent and child make. Each make is operating on its own. Make has no understanding of the “big picture”.

- When using build the developer supplies a buildfile in the parent directory that simply includes the buildfile in each of the subdirectories. This “parent” buildfile generally has a few simple rules stating that certain general targets depend on similar targets local to each of the subdirectories.

buildfile

```
SUB = subdir1 \
      subdir2

include optional rules $(SUB:=/buildfile)

all       : $(SUB:=/all)
clean     : $(SUB:=/clean)
install   : $(SUB:=/install)
```

By including all of the buildfiles the single build process has access to all of the rules and can construct a more complete view of the relationships – the dependencies. This allows build to understand the "big picture".

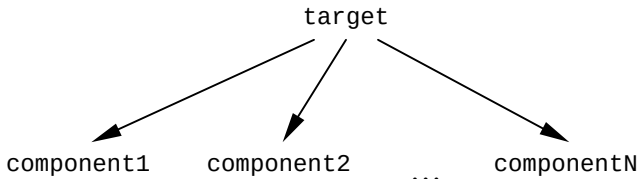
A number of make implementations support a simple include mechanism that does not provide for rule relocation or macro scope. Build's include directive supports a number of options that allow for better expression of the intent of the include operation. There are generally two reasons for including a file: to import macro definitions common to various parts of a complex project (shared definitions) and to import additional rules. The first is an example of a *simple* include, the text of the included file is processed as though it were part of the including file. This operation supported by both make and build. The second is unique to build and is how build deals with directory hierarchy, multiple products and encapsulation. By supporting the importation and relocation of rules build permits a complex project to be partitioned into manageable chunks.

Relationships

One of the essential features of both makefiles and buildfiles is the expression of dependency: Some target depends on a number of components. If the target doesn't exist or is "out-of-date" with respect with any of its components then this target must be remade. The commands associated with the target are executed to create or update the target. The target-component relationship forms a partial ordering of these objects. Note that any target may be a component of another target. When all of the various relationships are considered a directed acyclic graph may be constructed. The method for describing simple dependencies with build is similar to that used by make:

```
target       : component1 component2 … componentN
```

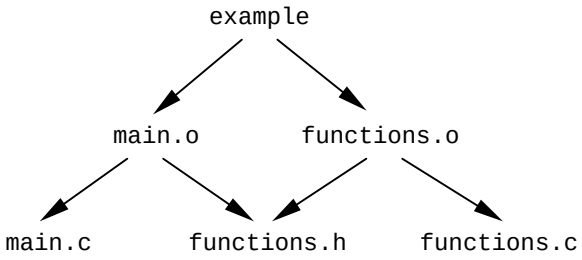
This simple line states that *target* depends on *component1*, *component2*, and so on. It represents the following directed acyclic graph:



Of course most applications require more than one rule. The following set of rules describes an example program formed from two source files and a common header file:

```
example     : main.o functions.o

main.o      : main.c functions.h

functions.o : functions.c functions.h
```

The above simple set of only three rules[1] has resulted in a dependency graph that is no longer a tree but a lattice:



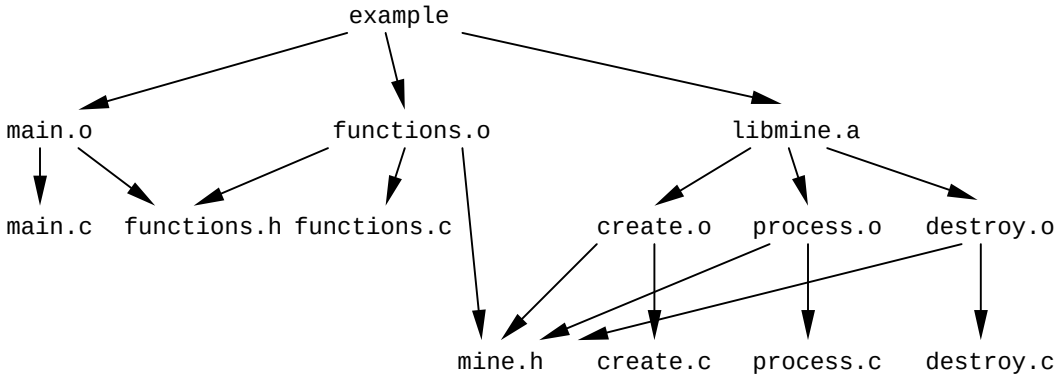
Note that in the above graph `example` is a *root* node as nothing depends on it. The objects `main.c`, `functions.c` and `functions.h` are *leaf* nodes as they depend on nothing. Finally main.o and functions.o are *interior* nodes. Unlike make, build understands the structure of the dependency graph or *lattice*. In general build recognizes that an object need only be made if it is out of date with respect to the leaves of its downset. Objects represented by interior nodes are intermediate and are not always needed. GNU make provides a directive to indicate that an interior node is an intermediate and need not be made unless some target to be made requires it. Build understands the intermediate nature of interior nodes directly from the structure of the lattice.

Use of libraries complicates things further by increasing the depth of the tree. As a first cut we'll just place the library source files in the same directory as the application and add rules for constructing the library to the application's buildfile:

```
example     : main.o functions.o libmine.a

main.o      : main.c functions.h

functions.o : functions.c functions.h mine.h

libmine.a   : create.o process.o destroy.o

create.o    : create.c mine.h

process.o   : process.c mine.h

destroy.o   : destroy.c mine.h
```

[1] The commands associated with the dependencies are not shown on many examples to save space.

This significantly complicates the dependency lattice:



As noted previously build will use its understanding of the dependency lattice to avoid doing work where possible. For example, suppose only the source files exist. Building *example* will result in everything being compiled and the library constructed and the link being performed. This completely populates the tree. Now suppose the source file *create.c* is modified. In order to reconstruct *example* build will only compile *create.c*, rebuild the library and relink *example*. Now if only *functions.h* is modified then build will only recompile *main.c* and *functions.c* and relink *example*. Finally if all the object files and the library were deleted and build instructed to reconstruct *example* (which is still there) then no work is done since build recognizes that *example* exists and is up-to-date with all its source files (the leaves of the graph).

Placing the library in the same directory as the application is a rather poor practice. The library should be placed in its own subdirectory. In addition it should have its own buildfile. The application buildfile:

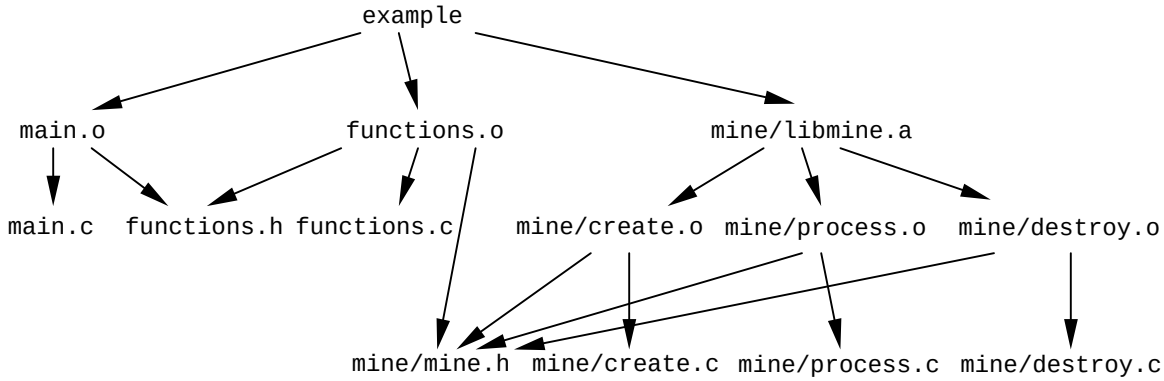
```
include rules mine/buildfile

example     : main.o functions.o mine/libmine.a

main.o      : main.c functions.h

functions.o : functions.c functions.h mine/mine.h
```

and the library buildfile (in the subdirectory):

```
libmine.a   : create.o process.o destroy.o

create.o    : create.c mine.h

process.o   : process.c mine.h

destroy.o   : destroy.c mine.h
```

Note that the library buildfile doesn't know or care what directory it is in. However the application directory has to know where to find the library's buildfile, the header file and the library file itself.

The structure of the dependency lattice is pretty much unchanged by this split, only the pathnames of the library objects are different:



The pathnames have changed because build recognizes that these objects are from the buildfile in the subdirectory, the rules were *relocated* appropriately. It should be noted that we don't have to keep the library directory as a subdirectory of the application. It is entirely possible that the library could become a product of its own. The only requirement is that the application buildfile must "know" where to find the files mine.h and libmine.a Generally such files are "installed" in some public place. The application need only include the library's buildfile if we want to maintain a developer relationship between the application and the library. The library's buildfile could be modified to allow for the installation of the header and library file:

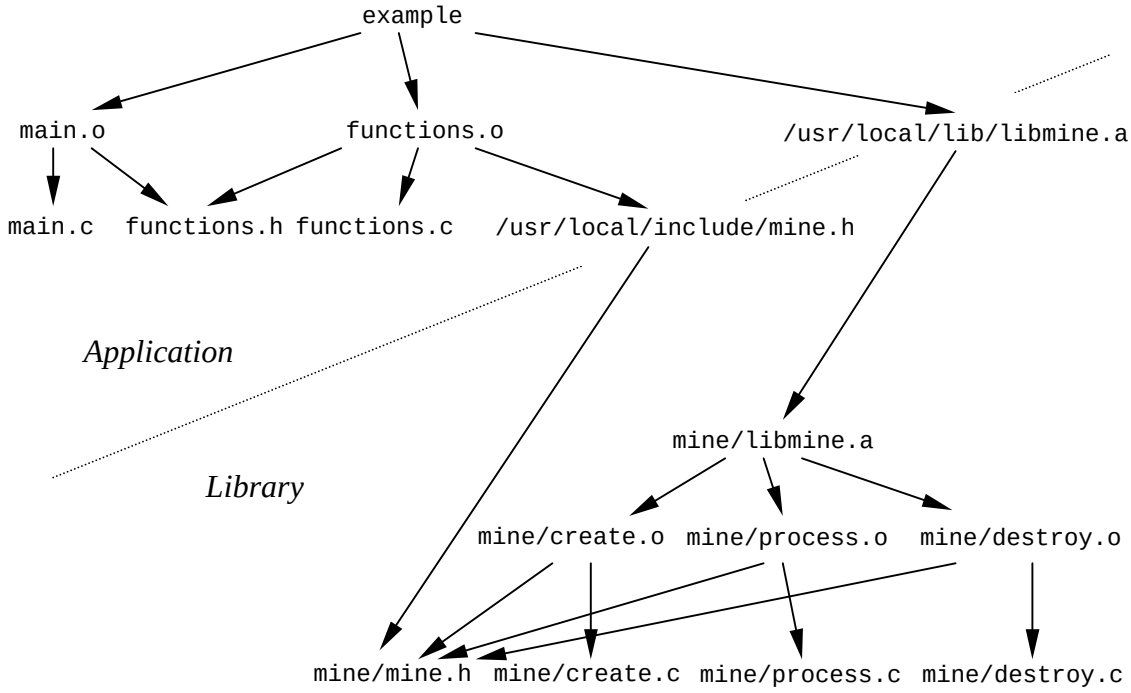
```
libmine.a   : create.o process.o destroy.o

create.o    : create.c mine.h

process.o   : process.c mine.h

destroy.o   : destroy.c mine.h

/usr/local/lib/libmine.a  : libmine.a

/usr/local/include/mine.h : mine.h
```

and the application buildfile modified appropriately:

```
include mine/buildfile

example     : main.o functions.o /usr/local/lib/libmine.a

main.o      : main.c functions.h

functions.o : functions.c functions.h /usr/local/include/mine.h
```

The change to the structure of the dependency lattice appears small but the ramifications are significant. The library has been well isolated from the application and may be moved freely. However build still has the big picture and can properly work with both the application and the library when it has the source and buildfiles for. In addition

build can properly deal with the application when only its source and buildfile are available.



Note that the lattice shown above has been effectively partitioned at the dotted line into two regions: the application objects and relations, and the library objects and relations (each region corresponds to a buildfile). The regions are joined at the installed objects: the header and the library files. The application region is a user of these and the library creates them. Of course we really should allow for the specification of where the library should be installed but this is really a matter of configuration scripting and build macro processing[2] and is a subject outside the scope of this paper.

Includes Relations

When a source file includes a header file there is an implied dependency. Consider the following simple C source file *eff.c*:

```
#include “eff.h”

int f ( int i )
{  return –i;
}
```

[2] Build’s macro processing support is generally a superset of the mechanism provided by GNU make.

The following makefile fragment supplies the relationship between the source file and an object file that is produced by compiling the source file:

```
eff.o         : eff.c
                cc –c eff.c
```

How does the header file fit in? The source file *eff.c* doesn't depend on the header file *eff.h*. Rather the object file depends on both *eff.c* and *eff.h*:

```
eff.o         : eff.c eff.h
                cc –c eff.c
```

This isn't too hard to do even if many source files include the header file. But suppose that someday the implementation of the header file *eff.h* is modified so that it includes some other header file *gee.h*. When make is used one would have to modify every rule with *eff.h* as a component adding *gee.h* as another component.

```
dee.o         : dee.c eff.h gee.h
                cc –c dee.c
 …

eff.o         : eff.c eff.h gee.h
                cc –c eff.c
```

Looking at the makefile one might expect to find that *dee.c* … and *eff.c* include both *eff.h* and *gee.h* but they don't. A developer might be tempted to "correct" the makefile to reflect what is in the source files, removing an important dependency. Note that the various source files such as *dee.c* and *eff.c* that include *eff.h* shouldn't have to "know" that *eff.h* happens to include *gee.h*. Similarly the rules for compiling these source files shouldn't need to know this either. It is a question of encapsulation. Suppose at some future date the implementation of *eff.h* is changed to include *cee.h* rather than *gee.h* and also include *why.h*. Having to track down and modify every rule that has *eff.h* as a component is unreasonable.

To address this sort of situation, build supports a mechanism for indicating such an includes relationship:

```
eff.h includes gee.h

dee.o         : dee.c eff.h
                cc –c dee.c
 …

eff.o         : eff.c eff.h
                cc –c eff.c
```

Now if the implementation of *eff.h* is changed to include *cee.h* rather than *gee.h* and also include *why.h*, then rather than modifying every rule with *eff.h* as a component, one need only change the single line to:

```
eff.h includes cee.h why.h
```

In reality, the statement that *dee.o* depends directly on both *dee.c* and *eff.h* isn't really correct. More accurately *dee.o* depends on *dee.c* and *dee.c* includes *eff.h*. So one could write a buildfile:

```
eff.h includes gee.h

dee.c includes eff.h

dee.o         : dee.c
              cc –c dee.c
 …

eff.c includes eff.h

eff.o         : eff.c
              cc –c eff.c
```

Note that in practice one would use a tool such as *builddepends* (like *makedepends*) that analyzes source files of popular languages to detect include operations and to generate an output file that contain the implied include relationships. One would then include this file from the buildfile. In summary, given:

```
A.h includes B.h C.h
```

If some target depends on *A.h* then it also depends on *B.h* and *C.h*. The process exhibits closure so given:

```
A.h includes B.h C.h
X.h includes A.h Y.h Z.h
```

If some target depends on *X.h* then it also depends on *A.h*, *B.h*, *C.h*, *Y.h*, and *Z.h*.

Parallelism

Build was designed from the start to support parallel operation. Several make implementations such as GNU make and pmake also support parallel operation to some degree. Note that for make or build, parallel operation doesn't mean constructing a parallel code, it means executing the commands associated with targets in a parallel fashion. Consider the following buildfile:

```
example     : main.o functions.o
              cc –o example main.o functions.o

main.o      : main.c functions.h
              cc –c main.c

functions.o : functions.c functions.h
              cc –c functions.c
```

The rule to produce *main.o* by compiling *main.c* and the rule producing *functions.o* by compiling functions.c may be performed in parallel. Both compiles may done simultaneously. This is because main.o doesn't depend on in any fashion, directly or indirectly, *functions.o* (nor does functions.o depend on main.o). Note that the link can not

be done in parallel with the compiles since the executable file depends on the object files and so can't be done until the object files have been constructed.

Both GNU make and pmake support parallelism in a single directory. If two or more targets are in the same directory and do not depend on each other then they may be constructed in parallel. They both support *in-directory* parallelism. Build supports general parallelism: if any targets do not depend on each other (directly or indirectly) then they may be constructed in parallel independent of directory. The only limit to parallelism for build is dependency and availability of processors.

Another issue is the proper recognition of targets. Consider a library formed from a large number of object files each the result of compiling a single source file. To keep things maintainable the library is assembled in a "chunkwise" fashion.

```
libgorfo.a   :: file_1_1.o file_1_2.o … file_1_M.o
             ar csr $@ $?

libgorfo.a   :: file_2_1.o file_2_2.o … file_2_M.o
             ar csr $@ $?

  …

libgorfo.a   :: file_N_1.o file_N_2.o … file_N_M.o
             ar csr $@ $?
```

Note the use of the colon-colon operator (`::`) in expressing the target-component dependencies. This is a special notation supported by both make and build that permits a given target to have multiple rules with commands. This is just what is needed in this case as the library is a single target, and multiple rules are used to assemble the library from subsets of the component object files.

This works just fine in serial with pretty much every implementation of make. However when operating in parallel both GNU make and pmake don't understand that the target found in each of the colon-colon rules is really the same target. For bookkeeping reasons, make appears to treat each as an independent target[3] and fair game for parallel operation. When executed in parallel the ar commands try to produce the same file simultaneously and the result is a corrupt library file[4].

Build treats colon-colon rules in a different fashion. Each is treated as a separate rule used to generate a single target by build. When a particular target is determined to be nonexistent or out-of-date with respect to its components then each rule associated with that target is performed one at a time in a serial fashion. Of course build will permit other independent operations to be performed in parallel with this now serialized stream of commands. In the end build properly supports colon-colon rules even when operating in parallel.

---

[3] Make really wants one rule with commands per target and fudges things internally to deal with the colon-colon rules.

[4] The work-around generally used by the ALE3D team is to build in parallel. When the link eventually fails due to one or more corrupt libraries the developer deletes all the libraries (leaving the object files) and re-runs the make in serial. The idea is to compile in parallel but assemble libraries and link in serial.

Pattern Rules

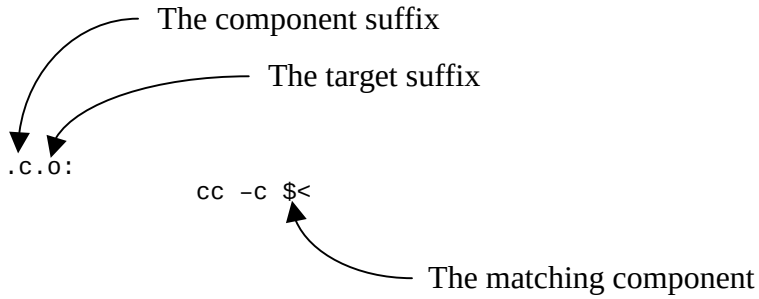
In addition to the support build provides for old-style make suffix rules, build supports two additional improved pattern rule mechanisms: one compatible with that supported by GNU make and a file-globbing pattern notation familiar to shell users.

Old-style Suffix Rules

Most make implementations support a mechanism for describing rules for making generic targets from generic components. Such rules are generally known as suffix rules. For example: a suffix rule describing how to compile a generic C source file to produce an object file could be:

```
.SUFFIXES: .c .o

.c.o:
            cc –c $<
```

While generally successful, few developers have been satisfied with the mechanism. The scheme is fairly limited to dealing with targets and components with specified suffixes (the file name extensions). There is no way of dealing with prefix-based rules or other interesting patterns. Worse yet, the description scheme is particularly non-intuitive. One has to use the special "`.SUFFIXES`" target to register the target and component suffixes. The suffix rule itself uses a special target assembled from the registered component and target suffixes, and has no specified components:



Note how the suffix rule notation has target and component reversed when compared to normal dependency notation (where target is on the left and components are on the right). Not all make implementations support a notation for a suffix rule for generating a target with no suffix (as is normal on UNIX-like systems). Some would support this as the following:

```
.o:
            cc –o $@ $<
```

Note how the special target only has the component suffix. This is fine for the typical situation where the expected target has no suffix and the component does, but what if the target is to have a suffix and the expected component does not?

GNU-style Pattern Rules

Like GNU make, build supports a *stem* notation system for describing how generic targets are to be remade from generic components. In this scheme, the special character % (percent sign) represents the parts of the names that match:

```
%.o : %.c
            cc –c $<
```

This notation is much more intuitive than the old style suffix rules as it follows the target and component conventions established for normal rules. In addition, this mechanism also supports prefixes:

```
% : s.%
            get $@
```

The above rule indicates how a file with any name may be derived from a file with the same name prefixed with s. using the *sccs* get command. Both GNU make and build support this form of pattern rule.

File-globbing Patterns

Users of most shells are familiar with wildcard-based patterns. These are formally known as *file-globbing* patterns. Build supports file-globbing patterns in pattern rules. For example a pattern rule for compiling a C source file:

```
*.o : $1.c
            cc –c $<
```

Note how the shell-like file-globbing pattern is used for the target on the left-hand side of the dependency. On the right-hand side a Perl-like dollar-digit notation is used that signifies the parts that matched. In the above example, the $1 is replaced with the string that matched the first wildcard (the star). This scheme can be used to supply rules for interesting (and potentially useful) situations. For example, the following pattern rule may be used to compile C source files that start with a vowel (a, e, i, o, or u):

```
[aeiou]*.o : $1$2.c
            cc –c $<
```

While the example is contrived, one can see that this scheme may be quite useful for defining pattern rules that compile some files (using particular options) and another pattern rule for compiling other files (with different options). The possibilities are intriguing. Reconsider the *sccs* example:

```
*.* : s.$1.$2
            get $@
```

File-globbing patterns are quite intuitive to developers on both UNIX- and Windows-like platforms. Build supports Bourne-shell compatible file-globbing patterns on all platforms.

Multi-Target Rules

Most commands used in rules generate a single file. For example: The C compiler produces a single object file as its output. This is easily illustrated by a rule such as:

```
banana.o    : banana.c
              cc –c banana.c
```

However a few commands produce multiple output files. For example: Parser generators such as yacc, bison, or slr typically generate a .c and a .h file. This may be shown in a buildfile by simply writing a rule with multiple targets such as:

```
parser.c parser.h : parser.pda
              slr parser.pda

parser.o    : parser.c parser.h
              cc –c parser.c

main.o      : main.c parser.h
              cc –c main.c
```

Many make implementations can't describe such a situation[5]. With these utilities the developer has to ignore one of the targets while using the other even with other rules. For example the above buildfile fragment would have to be written as:

```
parser.c    : parser.pda
              slr parser.pda

parser.o    : parser.c
              cc –c parser.c

main.o      : main.c parser.c
              cc –c main.c
```

The makefile doesn't reflect reality: *main.o* doesn't really depend on *parser.c*, it really depends on *parser.h*. The buildfile illustrates the real relationship.

Scalability

The local nature of buildfiles tends to support scalability. Each buildfile can generally be created and maintained independently. While the overall dependency lattice grows in complexity with each additional buildfile, the complexity of the individual buildfiles remains rather low. Each buildfile only needs to deal with is little piece of the problem, build will stitch them all together to form the big picture.

To reduce the amount of storage required, build goes to some length to ensure that each object (target or component file) is represented in the lattice by a single node. Build keeps track of what buildfiles have been processed to avoid revisiting buildfiles reached

[5] GNU make supports multi-target rules.

through different paths. In addition most of the algorithms used by build operate in a *O(n)* fashion (where *n* is the number of nodes in the lattice).

Timing results for building *totebag*

The benchmark used to test build was *totebag*, a complex meta-project with a rather large source directory tree containing a large number of products: libraries, applications and scripts. To construct and install everything build starts with a single top-level buildfile and through inclusion ends up processing 1160 buildfiles containing: 111308 macro definitions and 23382 rules, 75533 macros expansions are made (many nested). The dependency lattice contains 25043 nodes joined by 114067 edges with a maximum depth of 27 levels. This is clearly a BIG problem but build deals well with it. Most real applications of build are more reasonable.

The following table shows the amount of time build spends performing a number of internal activities. The two most interesting cases are shown: the "worst" case, where everything must be reconstructed, and the "best" case, where everything is up to date. Note that in the best case all intermediate objects where removed before the test. All performance testing was done on a 500 MHz Pentium 3 laptop with a 5400-RPM disk running Linux. All times in Table 1 are expressed in seconds.

Table 1. Performance by activity

	reading (sec)	processing (sec)	work (sec)	total (sec)
building everything	37.60	30.77	1477.48	1545.85
building nothing	37.39	24.23	0.00	61.62

The first column is the time that build takes to read, perform macro expansion, and construct the raw dependency lattice. The second column is the time build takes to analyze the lattice. This includes the time build takes to *stat* the various file system objects to determine if they exist and if they are up to date. The sum of first and second columns generally constitutes the *overhead* of the build process. The third column is the time taken by commands to bring things up to date.

As expected in the first row build had to go through the maximum amount of work and dispatch many commands. This is the worst case on time but best case for overhead ratio (overhead work vs. dispatched work). Note that the work column dominates the time taken. The second row is where everything has been constructed so build will no work to dispatch and is so dominated by columns one and two (overhead). This is the best case for overall time but gives poor overhead ratio, all of the time was spent on overhead work.

Performance vs. buildfile size

A number of individual products within the totebag meta-project have been processed with the intent of gaining some sort of understanding of how build's performance varies by the number of rules, objects, and relations[6]. The following figures illustrate the result of a series of 564 tests. Each test was arranged so that the desired product was already up to date, so all of the time spent is on overhead work[7]. The tests were chosen with problem sizes ranging over 4 orders of magnitude.

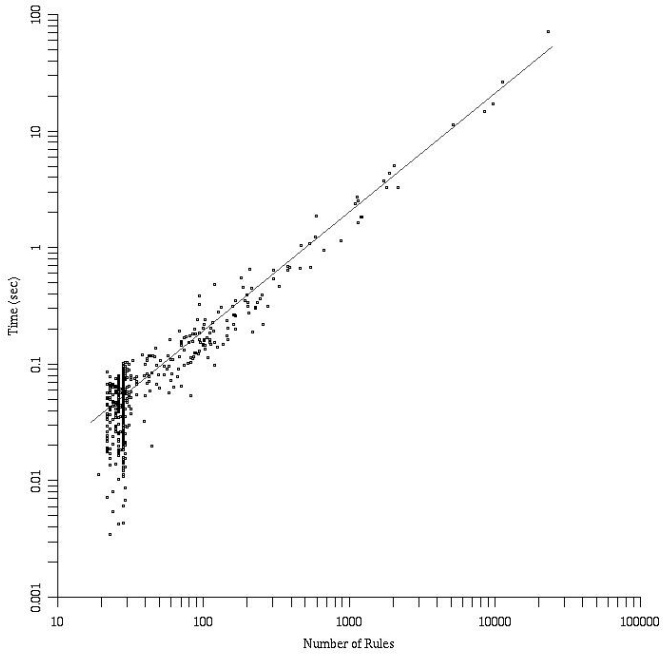


Fig 1. Log-log plot of *time* vs. number of *rules*.

Each of the plots was done in a log-log fashion so as to understand the exponent of the relationship between the factor (number of rules, objects, or relationships) and the time taken. Each point represents one of the many test problems. Note that in each of the plots the slope of the trend line is less than 1-to-1. This implies that the time performance may actually be slightly *sublinear* with the number of rules, objects, and relations. No claim is actually made for sublinear performance as timing inaccuracies are not easily predicted and the trend line fit isn't that accurate particularly at the low-end. But the overall behavior is approximately linear with problem size over a wide range.

---

[6] An object is a file, directory, or a phony such as *install* or *clean*; a vertex in the dependency lattice. A relationship is a dependency between two objects; a single edge in the dependency lattice.

[7] Having work to do such as compiles would mix compiler performance in to the test. The goal is to test build not the compiler.

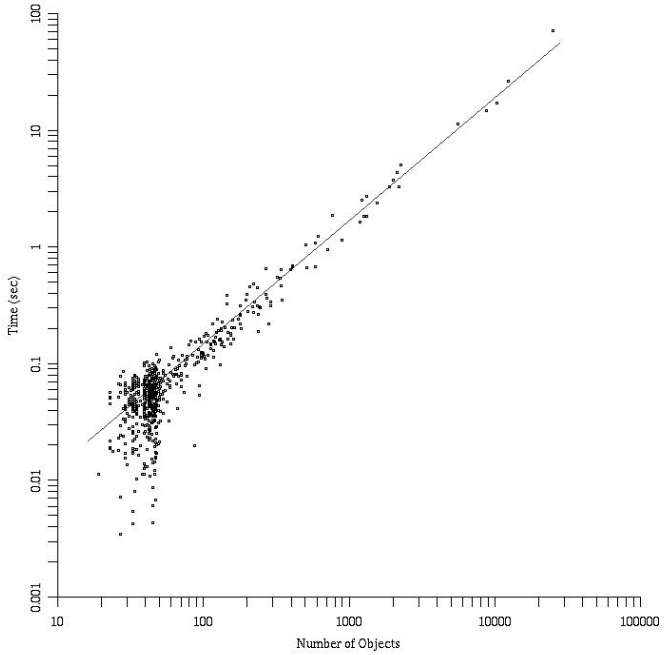


Fig 2. Log-log plot of *time* vs. number of *objects*.

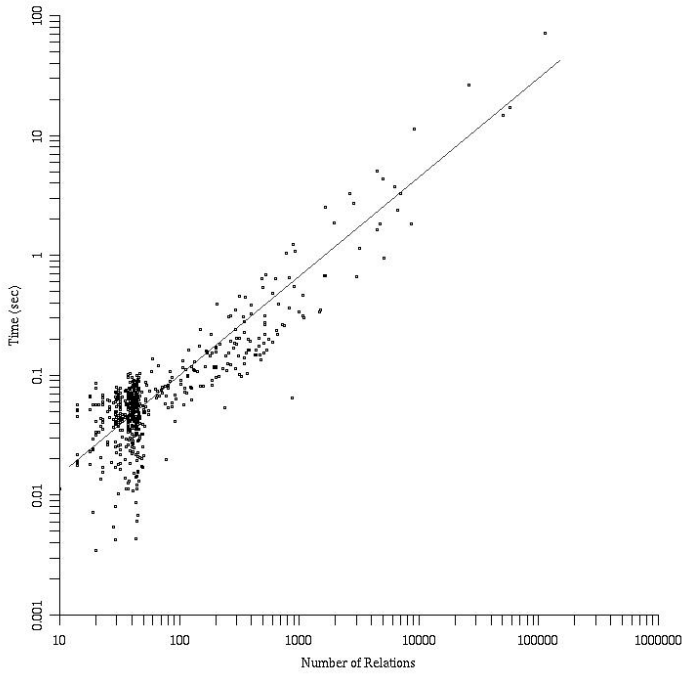


Fig 3. Log-log plot of *time* vs. number of *relations*.

Compatibility

Unlike a number of other make replacements, build is generally make compatible. Build was designed to be as compatible with make as possible. Buildfiles are very similar to makefiles. Build can read and process most well written makefiles. Build supports most of the GNU make extensions including GNU make's sophisticated macro processing extensions (macro functions). However build is not completely compatible with make:

- Build doesn't supply any built-in suffix rules. Build does support suffix rules but supplies no defaults. The buildfile developer must supply all needed suffix rules. This hasn't proved much of a problem as most makefiles that rely on suffix rules also supply their own definitions.
- A number of obscure macros such as $(@D), $(@F), … $(?F) are not currently supported. There are better ways of performing these operations so resolving this incompatibility has been a rather low priority. Implementation of these shouldn't be hard and will be supplied in the near future.
- Some System-V make features such as $$@ are not supported at this time.
- Chaining of implicit (suffix) rules isn't well supported. Improvements that provide proper support are being tested but are not ready yet.
- A few GNU macro functions are not supported yet:
    - The *call* function isn't supported.
    - The *eval* function isn't supported yet.
    - The *foreach* function isn't supported yet.
    - The *shell* function isn't supported yet.
    - The *value* function isn't supported yet. But it doesn't appear to work right in GNU make either.
    - The *wildcard* function isn't supported yet.
- Archive members. The notation: *archive*(*member* …) which was introduced in SunOS 4 make is not supported.
- Recursive builds work but are not recommended, as they are generally a poor use of resources. A better method is to learn about and use build's new include mechanism.

In general any developer familiar with make can use build with little additional training.

Portability

Build is configured with a script generated by *autokonf* [8]. It was written with portability in mind and can be built and used on a diverse set of platforms:

- AIX – 4.3, 5.1, and 5.2 (power)
- FreeBSD – 4.x and 5.4 (Intel)
- HP-UX – 9.05, 10.20, and 11.0 (PA-RISC)
- Linux – 2.2.x and 2.4.x (alpha and Intel), SuSE, Redhat, and Mandrake distributions.
- FreeBSD – 4.x and 5.4 (Intel)
- MAC OS-X (PowerPC)
- Windows – 98, NT 4.0, Win2000, and XP (Intel)
- SGI IRIX – 5.3 and 6.5 (mips)
- Tru64 – 3.2c, 4.0d, 4.0e, 5.0, 5.1, and 5.1b (alpha)

Build has been constructed, tested, and actually used on each of the above platforms. Note that neither Cygwin nor MKS environments are needed to construct build on Windows-like systems; the utility operates in a native fashion. Build has been constructed on the above platforms using the following C compilers:

- GNU (gcc 2.x and 3.x) on AIX, FreeBSD, HP-UX, alpha and Intel Linux, and Windows.
- HP native C compiler (cc) on HP-UX.
- HP/Compaq C compiler (cc) on Tru64 and alpha Linux
- IBM native C compiler (xlc) on AIX
- Intel C compiler (icc 7.x and 8.x) on Linux and Windows
- Microsoft Visual C (cl) on Windows
- Portland Group (pgcc) on Linux
- SGI native C compiler (cc) on IRIX

Currently an ANSI C compiler is required. It should be noted that few ANSI features are actually required beyond support for function prototypes. A C compiler that meets the 1989 standard is generally sufficient.

[8] *Autokonf* is a Perl-based replacement for GNU autoconf. It is implemented in Perl, uses macros written in Perl and generates a configuration script in Perl.

Conclusions

Build has been shown to be an adequate replacement for make. Aside from a few unresolved GNU make compatibility shortcomings, build is at least as capable as make on all counts, and demonstrates significant improvements over make on several key points:

- The ability to handle directory hierarchies.
- Encapsulation of projects with multiple applications and/or libraries.
- The automatic recognition of intermediate files and the ability to avoid dispatching of unnecessary commands (minimizing work where possible).
- Correct parallel operation.

In addition build has demonstrated good scalability as the performance varies linearly by problem size.

Future Work

While build works quite well and can meet the needs of most users there is still much work to be done:

- GNU make compatibility. The remaining features supported by GNU make but not by build should be implemented.
- Comparison of build's performance relative to make. Much testing was done to understand how build itself performs but no work has been done to understand how make performs when doing the same tests.
- Automatic derivation of dependencies. A first cut would be a *builddepends* utility in the same spirit as *makedepends*. An integrated mechanism would be more desirable but requires a great deal more thought.
- Support for distributed builds. Where the source is distributed across a number of possibly remote platforms. This needs a lot of thought even for defining what is really desirable.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

Eggert, Paul. Silogic Inc, Los Angeles, CA, private communication (1984)

Holt, Gary. *Makepp*. 14 Dec. 2004.
SourceForge. <http://makepp.sourceforge.net>.

Knight, Steven. *Scons*. 28 Sep. 2004.
The Scons Foundation.<http://www.scons.org>.

Schilling, Jörg. *Smake*. FhG Fokus.
<http://www.fokus.fraunhofer.de/research/cc/berlios/employees/joerg.schilling/private/smake.html>.

Stallman, R. and McGrath, R, *GNU make*. 28 Jun. 2002. Free Software Foundation. <http://www.gnu.org/software/make/manual/make.html>.

Oram, A., and Talbott, S., *Managing Projects with make*, (O'Reilly & Associates, 1993).

DuBois, Paul., *Software Portability with imake*, (O'Reilly & Associates, 1994).